Principles

Description

Provides information and data to educate software development professionals on the concept, applicability, and value of software security principles. It will also contain a set of key secure software principles to assist software development professionals in analyzing and creating their software architectures from a security perspective and in gaining a greater understanding of the key underlying concepts and patterns that, depending on how they are addressed, can make software either more or less secure.

Overview Articles

Name	Version Creation Time	Abstract
Design Principles	11/12/09 3:27:29 PM	As the recognition of security as a key dimension of high-quality software development has grown, the understanding of and ability to craft secure software has become a more common expectation of software developers. The challenge is in the learning curve. Most developers don't have the benefit of years and years of lessons learned that an expert in software security can call on. In an effort to bridge this gap, the Principles content area, along with the Guidelines and Coding Rules content areas, presents a set of practices derived from real-world experience that can help guide software developers in building more secure software.

Most Recently Updated Articles [Ordered by Last Modified Date]

Name	Version Creation Time	Abstract
Separation of Privilege	11/12/09 3:55:20 PM	A system should ensure that multiple conditions are met before granting permissions to an object. Checking access on only one condition may not be adequate for strong security. If an attacker is able to obtain one privilege but not a second, he or she may not be able to launch a successful attack. If a software system largely consists of one component, the idea of having multiple checks to access different components cannot be implemented. Compartmentalizing

		software into separate components that require multiple checks for access can inhibit an attack or potentially prevent an attacker from taking over an entire system.
Securing the Weakest Link	11/12/09 3:54:54 PM	Attackers are more likely to attack a weak spot in a software system than to penetrate a heavily fortified component. For example, some cryptographic algorithms can take many years to break, so attackers are not likely to attack encrypted information communicated in a network. Instead, the endpoints of communication (e.g., servers) may be much easier to attack. Knowing when the weak spots of a software application have been fortified can indicate to a software vendor whether the application is secure enough to be released.
Reluctance to Trust	11/12/09 3:54:33 PM	Developers should assume that the environment in which their system resides is insecure. Trust, whether it is in external systems, code, people, etc., should always be closely held and never loosely given. When building an application, software engineers should anticipate malformed input from unknown users. Even if users are known, they are susceptible to social engineering attacks, making them potential threats to a system. Also, no system is one hundred percent secure, so the interface between two systems should be secured. Minimizing the trust in other systems can increase the security of your application.
Psychological Acceptability	11/12/09 3:54:13 PM	Accessibility to resources should not be inhibited by security mechanisms. If security mechanisms hinder the usability or accessibility of resources, then users may opt to turn off those mechanisms. Where possible, security mechanisms should be transparent to the users of the system or at most introduce minimal obstruction. Security

		mechanisms should be user friendly to facilitate their use and understanding in a software application.
Promoting Privacy	11/12/09 3:37:22 PM	Protecting software systems from attackers that may obtain private information is an important part of software security. If an attacker breaks into a software system and steals private information about a vendor's customers, then their customers may lose their confidence in that software system. Attackers may also target sensitive system information that can supply an attacker with the details needed to attack that system. Preventing attackers from accessing private information or obscuring that information can alleviate the risk of information leakage.

All Articles [Ordered by Title]

Name	Version Creation Time	Abstract
Complete Mediation	11/12/09 3:19:45 PM	A software system that requires access checks to an object each time a subject requests access, especially for security-critical objects, decreases the chances of mistakenly giving elevated permissions to that subject. A system that checks the subject's permissions to an object only once can invite attackers to exploit that system. If the access control rights of a subject are decreased after the first time the rights are granted and the system does not check the next access to that object, then a permissions violation can occur. Caching permissions can increase the performance of a system, but at the cost of allowing secured objects to be accessed.
Defense in Depth	11/12/09 3:26:57 PM	Layering security defenses in an application can reduce the chance of a successful attack. Incorporating redundant security mechanisms requires an attacker to circumvent each mechanism to

		gain access to a digital asset. For example, a software system with authentication checks may prevent an attacker that has subverted a firewall. Defending an application with multiple layers can prevent a single point of failure that compromises the security of the application.
Design Principles	11/12/09 3:27:29 PM	As the recognition of security as a key dimension of high-quality software development has grown, the understanding of and ability to craft secure software has become a more common expectation of software developers. The challenge is in the learning curve. Most developers don't have the benefit of years and years of lessons learned that an expert in software security can call on. In an effort to bridge this gap, the Principles content area, along with the Guidelines and Coding Rules content areas, presents a set of practices derived from real-world experience that can help guide software developers in building more secure software.
Economy of Mechanism	11/12/09 3:27:57 PM	One factor in evaluating a system's security is its complexity. If the design, implementation, or security mechanisms are highly complex, then the likelihood of security vulnerabilities increases. Subtle problems in complex systems may be difficult to find, especially in copious amounts of code. For instance, analyzing the source code that is responsible for the normal execution of a functionality can be a difficult task, but checking for alternate behaviors in the remaining code that can achieve the same functionality can be even more difficult. One strategy for simplifying code is the use of choke points, where shared functionality reduces the amount of source code required for an operation. Simplifying design or code is not always easy, but

		developers should strive for implementing simpler systems when possible.
Failing Securely	11/12/09 3:28:28 PM	When a system fails, it should do so securely. This typically involves several things: secure defaults (default is to deny access); on failure undo changes and restore to a secure state; always check return values for failure; and in conditional code/ filters make sure that there is a default case that does the right thing. The confidentiality and integrity of a system should remain even though availability has been lost. Attackers must not be permitted to gain access rights to privileged objects during a failure that are normally inaccessible. Upon failing, a system that reveals sensitive information about the failure to potential attackers could supply additional knowledge for creating an attack. Determine what may occur when a system fails and be sure it does not threaten the system.
Least Common Mechanism	11/12/09 3:28:54 PM	Avoid having multiple subjects sharing mechanisms to grant access to a resource. For example, serving an application on the Internet allows both attackers and users to gain access to the application. Sensitive information can potentially be shared between the subjects via the mechanism. A different mechanism (or instantiation of a mechanism) for each subject or class of subjects can provide flexibility of access control among various users and prevent potential security violations that would otherwise occur if only one mechanism was implemented.
Least Privilege	11/12/09 3:29:15 PM	Only the minimum necessary rights should be assigned to a subject that requests access to a resource and should be in effect for the shortest duration

		necessary (remember to relinquish privileges). Granting permissions to a user beyond the scope of the necessary rights of an action can allow that user to obtain or change information in unwanted ways. Therefore, careful delegation of access rights can limit attackers from damaging a system.
Never Assuming That Your Secrets Are Safe	11/12/09 3:30:04 PM	Relying on an obscure design or implementation does not guarantee that a system is secured. You should always assume that an attacker can obtain enough information about your system to launch an attack. Tools such as decompilers and disassemblers allow attackers to obtain sensitive information that may be stored in binary files. Also, inside attacks, which may be accidental or malicious, can lead to security exploits. Using real protection mechanisms to secure sensitive information should be the ultimate means of protecting your secrets.
Promoting Privacy	11/12/09 3:37:22 PM	Protecting software systems from attackers that may obtain private information is an important part of software security. If an attacker breaks into a software system and steals private information about a vendor's customers, then their customers may lose their confidence in that software system. Attackers may also target sensitive system information that can supply an attacker with the details needed to attack that system. Preventing attackers from accessing private information or obscuring that information can alleviate the risk of information leakage.
Psychological Acceptability	11/12/09 3:54:13 PM	Accessibility to resources should not be inhibited by security mechanisms. If security mechanisms hinder the usability or accessibility of resources, then users may opt to turn off those mechanisms. Where possible, security mechanisms should

		be transparent to the users of the system or at most introduce minimal obstruction. Security mechanisms should be user friendly to facilitate their use and understanding in a software application.
Reluctance to Trust	11/12/09 3:54:33 PM	Developers should assume that the environment in which their system resides is insecure. Trust, whether it is in external systems, code, people, etc., should always be closely held and never loosely given. When building an application, software engineers should anticipate malformed input from unknown users. Even if users are known, they are susceptible to social engineering attacks, making them potential threats to a system. Also, no system is one hundred percent secure, so the interface between two systems should be secured. Minimizing the trust in other systems can increase the security of your application.
Securing the Weakest Link	11/12/09 3:54:54 PM	Attackers are more likely to attack a weak spot in a software system than to penetrate a heavily fortified component. For example, some cryptographic algorithms can take many years to break, so attackers are not likely to attack encrypted information communicated in a network. Instead, the endpoints of communication (e.g., servers) may be much easier to attack. Knowing when the weak spots of a software application have been fortified can indicate to a software vendor whether the application is secure enough to be released.
Separation of Privilege	11/12/09 3:55:20 PM	A system should ensure that multiple conditions are met before granting permissions to an object. Checking access on only one condition may not be adequate for strong security. If an attacker is able to obtain one privilege but not a second, he or she may not be able to

		launch a successful attack. If a software system largely consists of one component, the idea of having multiple checks to access different components cannot be implemented. Compartmentalizing software into separate components that require multiple checks for access can inhibit an attack or potentially prevent an attacker from taking over an entire system.